Announcements

- P6 due **Wednesday** at 11pm
- Final exam:
  - Thursday, 5/19, 9am, Barton Hall Indoor FieldWEST
- Please fill out two surveys online: course evaluation (College of Engrg) and experience in CS undergraduate courses (CS Dept). **Each will be worth 0.5 BONUS point.** Bonus point can be used against any point lost on the final exam (150 points).
- Regular office/consulting hours end Wednesday night. New hours to be posted online.
- Pick up papers during consulting hours at Carpenter
- Read announcements on course website!

- Previous Lecture:
  - Recursion – partitioning a triangle
  - Insertion Sort
  - (Read about Bubble Sort in Insight)

- Today’s Lecture:
  - “Divide and conquer” strategies
    - Binary search
    - Merge sort

---

**Insertion**

One insert process

Insert into sorted array of length 4

| 2 | 3 | 6 | 9 | 8 |
---|---|---|---|---|
| 2 | 3 | 6 | 8 | 9 |

Insert into sorted array of length 5

| 2 | 3 | 6 | 8 | 9 | 4 |
---|---|---|---|---|---|
| 2 | 3 | 6 | 8 | 9 | 4 |

Sort vector \( x \) using the **Insertion Sort** algorithm

Need to start with a sorted subvector. How do you find one?

\[
x \text{ Length 1 subvector is "sorted"}
\]

- Insert \( x(2): [x(1:2), C, S] = Insert(x(1:2)) \)
- Insert \( x(3): [x(1:3), C, S] = Insert(x(1:3)) \)
- Insert \( x(4): [x(1:4), C, S] = Insert(x(1:4)) \)
- Insert \( x(5): [x(1:5), C, S] = Insert(x(1:5)) \)
- Insert \( x(6): [x(1:6), C, S] = Insert(x(1:6)) \)

---

**Sort an array of objects**

- Given \( x \), a 1-d array of **Interval** references, sort \( x \) according to the widths of the **Intervals** from narrowest to widest
- Use the insertion sort algorithm
- How much of our code needs to be changed?

| A. No change |
| B. One statement |
| C. About half the code |
| D. Most of the code |

---

There are many algorithms for sorting

- **Insertion Sort** (to be discussed today)
- **Bubble Sort** (read Insight §8.2)
- **Merge Sort** (to be discussed next lecture)
- **Quick Sort** (a variant used by Matlab’s built-in sort function)

- Each has advantages and disadvantages. Some algorithms are faster (time-efficient) while others are memory-efficient
- **Great opportunity for learning how to analyze programs and algorithms!**
Searching for an item in an unorganized collection?

- May need to look through the whole collection to find the target item
- E.g., find value $x$ in vector $v$

```
v | | | | |
x
```

- Linear search

% Linear Search
% $f$ is index of first occurrence
% of value $x$ in vector $v$.
% $f = -1$ if $x$ not found.

```matlab
k = 1;
while $k <= \text{length}(v) \land v(k) \neq x$
    $k = k + 1;$
end
if $k > \text{length}(v)$
    $f = -1; % \text{signal for } x \text{ not found}$
else
    $f = k;$
end
```

Suppose another vector is twice as long as $v$. The expected “effort” required to do a linear search is ...

% Linear Search
% $f$ is index of first occurrence
% of value $x$ in vector $v$.
% $f = -1$ if $x$ not found.

```matlab
k = 1;
while $k <= \text{length}(v) \land v(k) \neq x$
    $k = k + 1;$
end
if $k > \text{length}(v)$
    $f = -1; % \text{signal for } x \text{ not found}$
else
    $f = k;$
end
```

An ordered (sorted) list

The Manhattan phone book has 1,000,000+ entries.

How is it possible to locate a name by examining just a tiny, tiny fraction of those entries?

Key idea of “phone book search”: repeated halving

To find the page containing Pat Reed’s number...

```
while (Phone book is longer than 1 page)
    Open to the middle page.
    if “Reed” comes before the first entry,
        Rip and throw away the 2nd half.
    else
        Rip and throw away the 1st half.
    end
end
```

What happens to the phone book length?

- Original: 3000 pages
- After 1 rip: 1500 pages
- After 2 rips: 750 pages
- After 3 rips: 375 pages
- After 4 rips: 188 pages
- After 5 rips: 94 pages
- After 12 rips: 1 page
Binary Search
Repeatedly halving the size of the “search space” is the main idea behind the method of binary search.

An item in a sorted array of length \( n \) can be located with just \( \log_2 n \) comparisons.

“Savings” is significant!

\[
\begin{array}{c|c}
 n & \log_2(n) \\
 100 & 7 \\
 1000 & 10 \\
 10000 & 13 \\
\end{array}
\]

% Linear Search
% f is index of first occurrence of value x in vector v.
% f is -1 if x not found.
\[
k=1;
while\ k<=\text{length}(v)\ && v(k)\neq x
\begin{align*}
&k= k + 1; \\
&\text{if}\ k>\text{length}(v)\ 
&f= -1; \ %\ \text{signal\ for\ x\ not\ found} \\
&\text{else} \\
&f= k;
\end{align*}
end
\]

Binary Search: target \( x = 70 \)

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
12 & 15 & 33 & 35 & 42 & 45 & 51 & 62 & 73 & 75 & 86 & 98 \\
\end{array}
\]

L: 6 \quad v(Mid) \leq x
Mid: 9 \quad So throw away the right half...
R: 12

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
12 & 15 & 33 & 35 & 42 & 45 & 51 & 62 & 73 & 75 & 86 & 98 \\
\end{array}
\]

L: 6 \quad x < v(Mid)
Mid: 9 \quad So throw away the right half...
R: 12

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
12 & 15 & 33 & 35 & 42 & 45 & 51 & 62 & 73 & 75 & 86 & 98 \\
\end{array}
\]

L: 6 \quad v(Mid) \leq x
Mid: 7 \quad So throw away the left half...
R: 9
Binary search: target $x = 70$

$$\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
v & 12 & 15 & 33 & 35 & 42 & 45 & 51 & 62 & 73 & 75 & 86 & 98 \\
\end{array}$$

L: 7  v(Mid) $\leq x$
Mid: 8  So throw away the left half...
R: 9

```matlab
function L = binarySearch(x, v)
    % Find position after which to insert x. v(1)<...<v(end).
    % L is the index such that v(L) $\leq x < v(L+1)$.
    % L=0 if x$v(1)$. If x>v(end), L=length(v) but x$\neq v(L)$.
    % Maintain a search window [L..R] such that v(L)$\leq x < v(R)$.
    % Since x may not be in v, initially set ...
    L=0;  R=length(v)+1;
    % Keep halving [L..R] until R-L is 1,
    % always keeping  v(L) $\leq x < v(R)
    % while R -> L+1
    m = floor((L+R)/2);  % middle of search window
    if  v(m) $\leq x$
        L=m;
    else
        R=m;
    end
end
```

function L = binarySearch(x, v)
    % Find position after which to insert x. v(1)<...<v(end).
    % L is the index such that v(L) $\leq x < v(L+1)$.
    % L=0 if x$v(1)$. If x>v(end), L=length(v) but x$\neq v(L)$.
    % Maintain a search window [L..R] such that v(L)$\leq x < v(R)$.
    % Since x may not be in v, initially set ...
    L=0;  R=length(v)+1;
    % Keep halving [L..R] until R-L is 1,
    % always keeping  v(L) $\leq x < v(R)
    % while R -> L+1
    m = floor((L+R)/2);  % middle of search window
    if  v(m) $\leq x$
        L=m;
    else
        R=m;
    end
end

 Done because  R-L = 1

Binary search is efficient, but we need to sort the vector in the first place so that we can use binary search

- Many different algorithms out there...
- We saw insertion sort (and read about bubble sort)
- Let’s look at merge sort
- An example of the “divide and conquer” approach using recursion

Which task is “easier,” sort a length 1000 array or merge two length 500 sorted arrays into one?

A. Sort  B. Merge

*Merge two sorted arrays so that the resultant array is sorted
Merge sort: Motivation

If I have two helpers, I'd...
• Give each helper half the array to sort.
• Then I get back the sorted subarrays and merge them.

What if those two helpers each had two sub-helpers?
And the sub-helpers each had two sub-sub-helpers? And...

Subdivide the sorting task

Subdivide again

And one last time

Now merge

function y = mergeSort(x)
% x is a vector. y is a vector
% consisting of the values in x
% sorted from smallest to largest.

n = length(x);
if n==1
    y = x;
else
    m  = floor(n/2);
    yL = mergeSortL(x(1:m));
    yR = mergeSortR(x(m+1:n));
    y  = merge(yL,yR);
end
The central sub-problem is the merging of two sorted arrays into one single sorted array.

```
function z = merge(x,y)
    nx = length(x); ny = length(y);
    z = zeros(1, nx+ny);
    ix = 1; iy = 1; iz = 1;
    while ix<=nx & iy<=ny
        if x(ix) <= y(iy)
            z(iz)= x(ix);  ix=ix+1;  iz=iz+1;
        else
            z(iz)= y(iy);  iy=iy+1;  iz=iz+1;
        end
    end
    while ix<=nx % copy remaining x-values
        z(iz)= x(ix);  ix=ix+1;  iz=iz+1;
    end
    while iy<=ny % copy remaining y-values
        z(iz)= y(iy);  iy=iy+1;  iz=iz+1;
    end
```

```
function y = mergeSort(x)
    % x is a vector.  y is a vector
    % consisting of the values in x
    % sorted from smallest to largest.
    n = length(x);
    if n==1
        y = x;
    else
        m  = floor(n/2);
        yL = mergeSort(x(1:m));
        yR = mergeSort(x(m+1:n));
        y  = merge(yL,yR);
    end

function y=mergeSort(x)
    n=length(x);
    if n==1
        y=x;
    else
        m=floor(n/2);
        yL=mergeSort(x(1:m));
        yR=mergeSort(x(m+1:n));
        y=merge(yL,yR);
    end
```

All the comparisons between vector values are done in **merge**.
How do merge sort and insertion sort compare?

- Insertion sort: (worst case) makes $i$ comparisons to insert an element in a sorted array of $i$ elements. For an array of length $N$: $1+2+...+(N-1) = N(N-1)/2$, say $N^2$ for big $N$
- Merge sort: $N \cdot \log_2(N)$
- Insertion sort is done in-place; merge sort (recursion) requires much more memory

See compareInsertMerge.m

What we learned...

- Develop/implement algorithms for problems
- Develop programming skills
  - Design, implement, document, test, and debug
- Programming "tool bag"
  - Functions for reducing redundancy
  - Control flow (if-else; loops)
  - Recursion
  - Data structures
  - Graphics
  - File handling

What we learned... (cont'd)

- Applications and concepts
  - Image processing
  - Object-oriented programming
  - Sorting and searching—you should know the algorithms covered
  - Divide-and-conquer strategies
  - Approximation and error
  - Simulation
  - Computational effort and efficiency

Computing gives us insight into a problem

- Computing is not about getting one answer!
- We build models and write programs so that we can "play" with the models and programs, learning—gaining insights—as we vary the parameters and assumptions
- Good models require domain-specific knowledge (and experience)
- Good programs ...
  - are modular and cleanly organized
  - are well-documented
  - use appropriate data structures and algorithms
  - are reasonably efficient in time and memory

Final Exam

- Thursday 5/19, 9-11:30am, Barton Hall indoor tracks WEST
- Covers entire course; some emphasis on material after Prelim 2
- Closed-book exam, no calculators
- Bring student ID card

- Check for announcements on webpage:
  - Study break office/consulting hours
  - Review session time and location
  - Review questions
  - List of potentially useful functions